2

SDRL Q14-02021-D

Q14 - Standards Development Plan Ada Interfaces to X Window System Analysis and Recommendations

Prepared for
Software Technology for Adaptable
Reliable Systems (STARS)
Unisys STARSCenter
Shipboard and Ground Systems Group
Reston, Virginia

Prepared by
Unisys Corporation
Defense Systems
System Development Group
P.O. Box 517, Paoli, PA 19301-0517

Contract No. F19628-88-D-0031
IDWO P.O. 010412

March 20, 1989

1 Executive Summary

This report describes the X Window System and its role in providing portable user interfaces for the bit-mapped graphics workstation market. It also shows how the Ada community can best take advantage of X to create portable applications and encourage code reuse. Included is a critique of a set of Ada bindings to X, and recommendations for a better approach for providing X for Ada applications.

The X Window System has become the standard window system for the workstation market. The Ada community can achieve many of its portability goals by leveraging the work done in the X Window System. Ada needs an implementation of the Xt toolkit intrinsics, as this is becoming the standard intrinsics for X toolkits, and needs a single set of user interface objects for consistent user interfaces among applications. The advances being made in user interface management system technology can be incorporated into an Ada user interface standard to provide additional portability and reusability. [illegible]

2 Introduction

Two major thrusts of the STARS program, and industry as a whole, are application portability and reuse of software components. Application portability can be divided into two types of portability; program portability, and end user portability. Program portability means a program written for one machine should require little or no modification to run on another machine. Program portability is a major concern of the Department of Defense (DOD) since the DOD uses a wide variety of hardware platforms. The move to a single programming language Ada by the DOD has opened the door to program portability. The second type of portability, end user portability, means a user should be able to run and interact with the same program on many different machines. The "look and feel" of a program should not change from host to host. Related to end user portability is consistent user interfaces among related programs. This is especially important to the STARS SEE where many software engineering tools from a variety of vendors must be integrated to produce a usable software development environment.

Studies [Myers 88] have shown that 29% to 88% of an application's code directly involves the user interface. Porting the user interface becomes a major task in porting an application to a new host. A common, portable user interface can greatly increase the portability of an application, aids user portability by keeping the the user interface constant on different hosts, and can improve cross-application user interface consistency.

A common user interface, usually in the form of subroutine libraries, makes a large body of reusable code available to the application development effort. User interface subroutines handle the details of formatting messages to and input from the user, thus an application developer need only be concerned with the interface to the user interface subroutines and not the details of handling data to and from the screen.

The intent of this paper is to address the need of STARS for a standard user interface, an issue that spans the whole computing community. Currently, the C community is addressing the same issue, and Ada can build on their efforts. The X Window System developed at M.I.T. is an established vehicle for standard user interfaces. This paper will describe the efforts to standardize X, and show how X supports application portability and reusability. In doing so the paper will show where Ada can take advantage of this work to fulfill its goals for application portability and software reusability.

This paper first briefly describes the X Window System, and shows how the layers of X contribute to portability and reusability. Included in the discussion of the X Window System layers are user interface management systems (UIMS), some built on top of X. The paper then looks at the current standardization efforts in organized standards bodies such as ANSI and IEEE, and in industry consortiums to show the directions X is taking and the opportunities for Ada to utilize this work. X is not the only window system, and alternative systems are discussed in the paper to show the strengths and weaknesses of X. Finally, the paper addresses the issue of what direction an Ada window system should take. The paper evaluates bindings/implementations of X in Ada, and from the lessons learned maps a strategy for an Ada implementation of an X based user interface system. The appendix presents design issues in implementing an Xt toolkit in Ada.

3 X Background

The X Window System [Scheifler 86] [OReilly 88] was developed jointly by M.I.T. and Digital Equipment Corporation (DEC) under Project Athena. The principal designers were Robert Scheifler of M.I.T. and Jim Gettys of DEC. The first public release was Version 10 Release 4 in 1986. In September 1987 a new release, Version 11 Release 1, occurred which was incompatible with the previous release. In March 1988 Version 11 Release 2 and in October 1988 Version 11 Release 3 were released, and were compatible with previous releases. Future releases, if any, are expected to be upward compatible. The release consists of source code, documentation, and sample programs.

By making source code available, M.I.T. has encouraged the proliferation of X. Many companies are now producing X implementations and X based applications. Releasing source, as M.I.T. did, can often lead to many distinct versions of X and actually prohibit portability, but to avoid this the X developers formed the X Consortium, consisting of industry and academic members, who define an official version of X, and control changes to X. A test suite is under development for verifying compliance with the X Consortium definition of X.

X is a windowing system for bit-mapped graphics displays in a network environment. X can display to multiple screens and accept input from a keyboard and a pointing device, usually a mouse. X consists of a server residing on a display device, such as a Sun or DEC workstation, a network protocol for communicating with application programs (clients) which may run anywhere on the network. Current implementations of X use TCP/IP and DECnet networks and UNIX domain sockets. The server manages the screen, allowing multiple applications to use the screen simultaneously. The server captures user input and directs the input to the correct application, supports two-dimensional graphics, and manages shared resources among clients, including windows, cursors, and fonts, and data structures such as graphics contexts for drawing to the display. To promote program portability and ease of programming, X uses a window manager to handle screen layout. A user can suggest screen layout, but the final decision rests with the window manager. An application need not be concerned with the characteristics of a particular display device a program might run on. Window managers are special X applications that are dependent on display device characteristics, and are shared by all applications on a device.

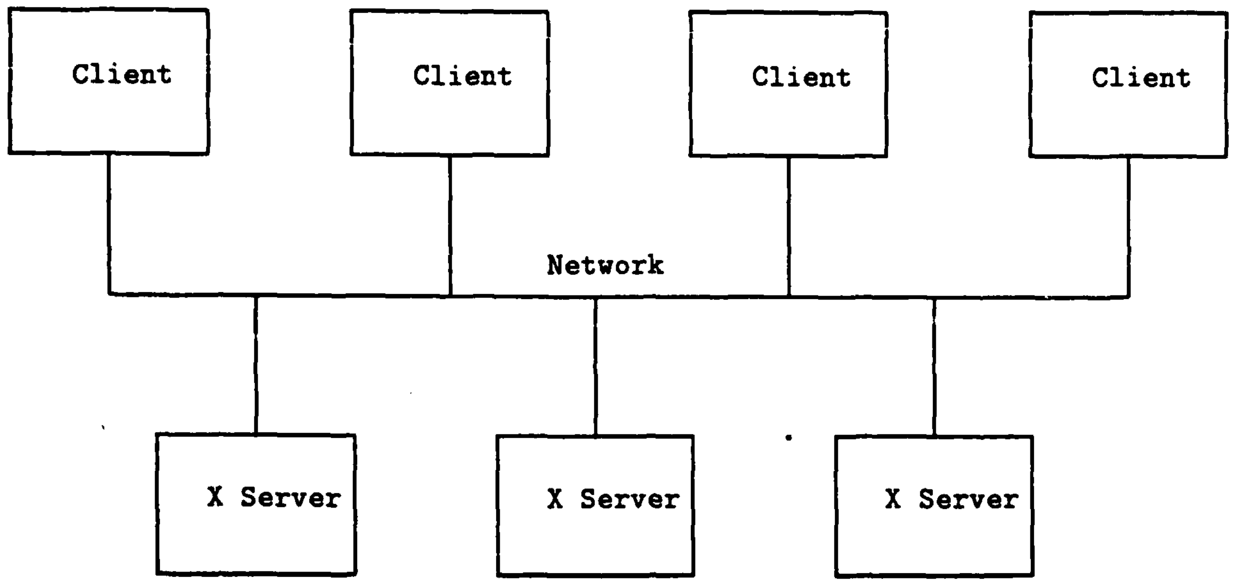

Figure 1: X Window System Architecture

The mechanism used by the server to pass information to the client is an event queue. Each application has a queue upon which the server places input to the application. The applications may also place events on its own queue and on other client queues. The server thus acts as an interprocess communication (IPC) vehicle, although the messages are limited to strings.

3.1 X Structure

The X Window System consists of several layers:

- X11 protocol
- Xlib
- intrinsics
- toolkits

The lowest layer is the protocol used to communicate between server and client. Above the protocol is Xlib the programming interface to the protocol. Xlib consists of over two hundred function calls to perform functions such as drawing to the screen, creating and deleting windows, selecting fonts and cursors, handling events, and setting and inquiring about window attributes such as foreground, background, and color. Xlib is the most primitive interface to X, but provides a sufficient set of functions for fully portable user interfaces. Applications written at the Xlib level

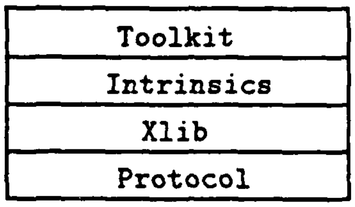

Figure 2: X Layers

require considerable programming effort since no basic objects such as menus and scrollbars are available, and thus contributes little to reusability.

The protocol and Xlib are the core of all X Window System implementations. Above the core several distinct paths exist. This paper looks at two of these paths: the X Toolkit or Xt from M.I.T., and the Xrlib Toolkit or Xr from Hewlett-Packard (HP). Both implementations consist of a set of intrinsics for building the higher level user interface components and libraries of components such as buttons, menus, and scrollbars. In Xt the intrinsics are also the application program interface to the toolkit. Figure 2 shows the relationship of the layers of X.

Xt is by far the most popular and many vendors, including DEC and HP, have extended Xt in their own products. The Xr toolkit usage is declining due mostly to the popularity of Xt. Ada bindings to the Xr toolkit exist, and this paper discusses the strengths and weaknesses of Xr as a result of working with the Ada bindings to Xr. No public domain Ada bindings to Xt exist at this time although there is work in progress. A few proprietary Ada bindings to Xt are rumored to exist.

3.2 X Toolkit - Xt

Xt [McCormack 88] [Burleigh 88] is an object-oriented toolkit used to build the higher level user interface components. The Xt Intrinsics provide primitive classes of objects and mechanisms to build, manage, and integrate more complex user interface objects. An object providing a user interface abstraction in Xt is a widget [Swick 88] which consists of an X window, window attributes, operations, and state information. The intrinsics provide the programmatic interface for both the widget programmer and the application programmer. The intrinsics provide routines for application initialization, widget creation and management, event handling, window geometry management, and resource management. Applications instantiate widgets which are maintained in a widget tree and provide functions which are "called back" when an event occurs within a widget. Applications generally consist of code to instantiate the widget instances for the application via intrinsics function calls, a set of callback functions for handling widget events, and a call to an intrinsics provided event dispatch function (an infinite loop pulling events off the client event queue and dispatching callback functions). Figure 3 shows the relationship among the various X layers and the application program.

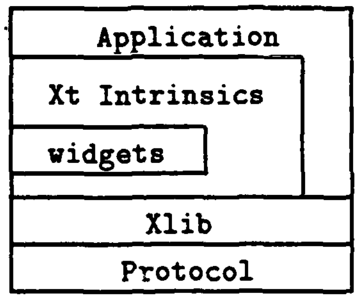

Figure 3: Application Program View of Xt

3.3 Xrlib Toolkit - Xr

Xr [HewlettPackard 88] consists of a set of intrinsic functions, a set of extensible field editors, and a set of dialog boxes. The intrinsics provide functions for building field editors, andling events associated with field editors and functions for combining field editors into dialog ooxes. The field editors are the basic building blocks of the user interface. Examples of field editors are scrollbars, button boxes (radio, pushbutton, check boxes, etc.), text editors for displaying and entering data including raster images, and titlebars. The field editors handle local events pertinent only to the field editor. For instance, a text editor for entering data will handle all keystrokes until the text input is complete and then notify the application that text has been entered and make the text input available to the application. The dialog boxes are higher level user interface components and usually encapsulate several field editors needed for a single user interface function. Xr comes with three dialog boxes:

- a message box for displaying brief messages to the user and a set of buttons for reply
- a menu dialog for building and handling menu operations
- a more general purpose panel dialog which permits the application to group together a complex combination of field editors for displaying and entering data

A panel dialog may have several text editors for entering user data and a row of buttons or check boxes for selecting options. Xr provides a mechanism for building new field editors which can be easily integrated into panel dialogs. Figure 4 shows the relationship among the various Xr layers and the application program.

4 X Layers - Portability and Reusability

This section describes the effect on portability and reusability of each of the layers of X, and includes a section on UIMS contributions to portability.

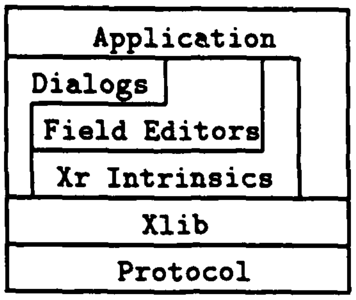

Figure 4: Application Program View of Xr

4.1 Protocol and Xlib

The protocol and Xlib layers provide the basic necessities for achieving application portability. By incorporating the user interface into Xlib calls an application's user interface can be easily ported to different hardware platforms. Because Xlib is so primitive these two layers provide little support for reusability. Even such basic structures such as menus and scrollbars must be built within the application. An application program must handle and interpret the individual keystrokes of user input. Unless a developer makes a conscious effort to separate the user interface components in his application the user interface becomes tightly bound to the application code, and, thus, difficult to reuse and develop consistent user interfaces across applications.

4.2 Intrinsics

The intrinsic layer provides the building blocks for creating reusable user interface components, where the toolkit layer provides the reusable components. In Xt the widget classes are the reusable components. These may be directly reusable as in menus and scrollbars or existing widgets may be modified or extended by using the inheritance mechanism provided in the object oriented approach of Xt. This method of inheritance is a fundamental concept in promoting reusable code. In Xr, the field editors and the dialogs are the reusable components. Xr does not have the concept of subclasses, and building new components from old components requires code modification.

The Xt toolkit provided by M.I.T. does not provide a large set of reusable interface components. The widget set, Athena widgets (really a sample widget set), provides only some basic building blocks for more complex widgets. The Athena widget set includes a command button widget, a scrollbar widget, label widget, and several types of forms and box widgets. An application using these widgets must do a considerable amount of work to create usable user interface components. Because of this many vendors have created there own widget sets using the Xt Intrinsics. With vendors developing their own widgets sets program portability, user portability and user interface consistency may suffer, thus portable applications on X require a single standard widget set.

The Xr toolkit provides many resuable user interface components with its field editors and dialogs. In Xr the application may have to handle much of the event handling which can reduce the reusability of code, but a mechanism is available for dispatching event handler function to specific windows.

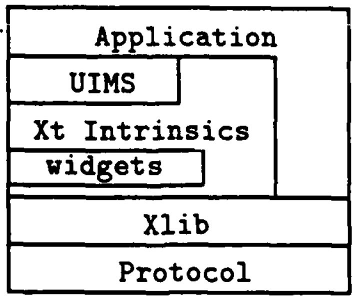

Figure 5: Application Program View of UIMS

The dialogs take advantage of this feature to handle events in various field editors contained in a panel.

4.3 UIMS

Another approach to user interfaces is user interface management systems or UIMS [Myers 88] [Lowgren 88]. A UIMS attempts to separate the user interface portion of an application from the application itself. This can help build better and more consistent user interfaces which are less dependent on the application semantics. To achieve application portability a UIMS must be built on top of a portable toolkit, otherwise the job of porting an application becomes a job of porting the UIMS tool. More UIMS systems are now being built upon X. OPEN/Dialogue from Apollo, TAE Plus from NASA [NASA 88], Serpent from Software Engineering Institute [Institute 88], and Chiron from University of California Irvine [Young 88] [Durand 89] are built upon X. The latter two generate user interfaces for Ada.

Using a UIMS introduces new problems into building good user interfaces. Some UIMS tools are difficult to use; the interface must be specified in a special language in some UIMS. Of great concern is the restriction in functionality often introduced by a UIMS. Many UIMSs "rarely can be used to help control the display and manipulation of the real application data objects [Myers 88]." UIMS technology is a very active research, and new, better products are coming out. With more experience the technology will mature and provide portable user interface support. Figure 5. shows the application program relationship to a UIMS built on top of Xt.

5 X Standardization Efforts

The X Window System has become a *de facto* standard. With the distribution of source code, a stable well-defined X was necessary, so M.I.T. organized the X Consortium in 1988 to continue support for X and provide some control over X. Any company or institution interested in contributing to X may become a member. Changes to the official release of X must be approved by the consortium. Currently, a test suite is under development for verifying compliance with the official X Consortium definition of X.

A group of UNIX vendors formed the Open Software Foundation (OSF) in 1988 to develop a single UNIX operating system and portable software. One task of OSF was to develop a plan for portable user environment. In a six month evaluation of existing vendor products, OSF developed an X Window System based user environment, called Motif[Hinckley 89], defined by combining HP's Common X Interface and Window Manager, and DEC's X User Interface and User Interface Language. Also included were the HP/Microsoft Style Guide which includes Presentation Manager compatibility. This forms the core of the OSF user environment. OSF also identified several UIMSs for OSF compliant environments:

- Open Dialog from Apollo
- BASE/OPEN from Swedish Telecom
- Generic Window Manager from Groupe Bull

Use of a UIMS is optional in OSF compliant environments. OSF plans to do further research on Carnegie Mellon's Andrew System, an object oriented user environment ported to work on X.

Sun Microsystems and AT&T agreed to merge their versions of UNIX, and have further agreed to use the OPEN LOOK Graphical User Interface which defines a user interface "look and feel." Sun's next generation toolkit View2 is similar to the original SunView, but implemented directly on Xlib. AT&T's XT+ is based on Xt intrinsics. Both adhere to the OPEN LOOK definition.

In addition to the industry cooperative agreements, formal standardization of X has begun. The ANSI graphics standards committee X3H3 has a task group X3H3.6 working on a formal standard for the X protocol. A draft is currently before X3H3 for ballot, and a standard is expected in 1990. Work should begin shortly on an Xlib standard in X3H3.6. Plans are to start with a C language binding for Xlib and follow with an Ada binding.

IEEE recently formed a committee P1201 to work on drivability standards and program interface standards for the upper layers of X. Drivability standards are an attempt to standardize application program actions for certain user actions, e.g. a standard action for a left mouse button click. OSF has taken the lead in defining the intrinsics and toolkit standards by offering Motif as the basis for P1201 work.

The National Institute for Standards and Technology (NIST, formerly NBS) has drafted a Federal Information Processing Standard (FIPS) for user interfaces. This FIPS is based on X and the Xt intrinsics,

The industry has embraced X for standard user interfaces, and all standards and cooperative efforts are Xt based. Clearly, an Xt based window system provides the greatest opportunity for portable C applications in the workstation environment. Ada applications should be able to capitalize on the directions provided by the efforts in C. Both the ANSI and IEEE standards bodies plan to offer Ada standards, and these will be based on functional specifications used in developing C standards including Xt intrinsics.

Model Layer	Component
6 Application	Application Logic
5 Dialogue	UIMS, Window Manager
4 Presentation	UIMS, Window Manager
3 Toolkit	Toolkit
2 Subroutine Foundation	Xt Intrinsics
1 Data Stream Interface	Xlib
0 Data Stream Encoding	X Protocol

Figure 6: NIST User Interface Reference Model

6 Alternative Window Systems

X is not the only window system, and competition from other window systems remains a barrier to application program portability. Most of these competing windowing systems are proprietary which precludes their replacing X. Principal reasons for X's popularity are that X is not proprietary, M.I.T. encourages porting X to different hosts, and the widespread desire for application portability. The Apple Macintosh window system might be a good candidate for a standard window system, but Apple has made it clear that it intends to keep its software proprietary. Sun Microsystem's Network Extensible Window System NeWS came too late to compete with X, and Sun's window system now supports both X and NeWS.

The most serious competitor might be IBM's Presentation Manager. The large OS/2 market makes Presentation Manager a desirable window system since a large application base exists for PCs which vendors would like to port to other hosts. IBM is moving to make Presentation Manager a window interface standard [Hanner 88]. Both X and Presentation Manager are moving into each others domain. Work is underway in various companies to put X on PCs. Currently, performance problems and memory consumption are a problem in running X on PC class machines. HP is attempting to put Presentation Manager on top of X and permit both X and Presentation Manager to exist simultaneously on a system.

X and Presentation Manager have strengths; X in the distributed UNIX environment and Presentation Manager its rich application program interface (API) and large market. Both have weaknesses; X has many different toolkits, and Presentation Manager does not address a distributed environment. X does not yet address the large body of UNIX users using ASCII terminals, although Visual Technology is developing an X terminal to address this market. See [Morris 89] for an extended discussion. The standardization efforts by X Consortium, OSF, ANSI and IEEE may resolve the variety of APIs currently on the market for X, but not soon. The use of the Presentation Manager Style Guide by OSF may provide common "look and feel" between the two window systems. Over time the two window systems may converge into a single window system or at least develop common APIs which would solve the portability problems, but nothing can be expected soon, and no clear movement is apparent to merge the two.

7 X and Ada

Most activity involving X is in the C community, but there is a small group of people working on X Ada bindings or implementations. Science Applications International Corporation (SAIC), in a STARS Foundation contract, produced public domain Ada bindings for Xlib and the Xr toolkit, and is also working on an Xt Ada implementation. There exist some proprietary bindings or implementations of Xt, but these are not well publicized or available for evaluation. Sanders Associates is working on an Ada X server which they will consider adding to the public domain. Work on an Ada version of X by the X Consortium has been proposed by its director Robert Scheifler, but as yet no companies have shown enough interest to fund the work. In the ANSI and IEEE communities there is only a small interest in Ada bindings. Both X3H3.6 and P1201 have committed first to developing C bindings leaving work on Ada bindings for later. There are two UIMSs that generate X code in Ada, TAE Plus and Chiron. Neither are product quality at this point.

The STARS Q14 task has worked with the SAIC bindings to Xlib and Xr, and the remainder of this section presents observations and conclusions on the usefulness of these bindings, and suggestions for future directions.

7.1 Ada Xlib Bindings

The SAIC Xlib bindings are by far the best of the two X bindings. The SAIC bindings are a shallow interface to the X C code relying on direct calls to the C functions. The Unisys STARS Standard Interfaces task found a few problems in the SAIC implementation, but none serious and all were easily corrected. Missing from the SAIC bindings are a few Xlib functions which require procedure variables as parameters to function calls. Most of the missing Xlib functions permit applications to dynamically assign protocol error handling routines by passing procedure variables as parameters to functions. Ada does not directly support procedure variables, but methods exist for supporting the functionality [Lamb 83]. However, these mechanisms would not work in a shallow interface implementation.

A shortcoming in the Xlib bindings is the representation of event types as enumerated types. In the C version the events are represented as integers with a large block of consecutive integers, beginning with zero, reserved by the X Consortium for future use. X was designed to be easily extensible, but by using enumerated types for event types, adding new events is nontrivial. An Ada application could not create a new event type without modifying the the Xlib bindings. To abide by the X Consortium conventions would mean adding a large number, thousands, of unused event types to ensure the position in the enumerated type declaration matches the event numbers used in the C code. The Xr toolkit actually has a new event type, and rather than alter the types in Xlib, new routines for reading the event queue are added to Xr, but the Ada Xlib functions to read the event queue can not handle events with this new event type. Enumeration types for events limit the extensibility of X, an important feature of X.

Otherwise, the SAIC bindings are a good starting point for an Ada API to the X protocol. The ANSI X3H3.6 work on Xlib Ada bindings will use the SAIC bindings as a starting point for its standards work, and by adding the missing functionality and correcting the event typing problem, a good Xlib binding will be available to Ada applications.

7.2 Ada Xr Bindings

The Ada Xr bindings, also written by SAIC under their STARS Foundation contract, are also a shallow interface relying on direct calls to the HP Xr C code, but unlike the Ada Xlib bindings are much less mature. There are many implementation errors; most not difficult to fix. Testing seems to have been minimal since many of the bugs render a particular function completely unusable. With some perseverance a reasonable application can be prototyped. The current version is certainly not of production quality. Honeywell is currently doing a thorough testing and debugging of the code, and once that is completed a more usable version should be available in the public domain. Even with the Honeywell work, Ada Xr has serious problems. Firstly, one important feature of C Xr is missing. Xr offers the ability to create window handlers and menu handlers which are passed as procedure variables to the C Xr code. As in Xlib, features requiring procedure variables are omitted. Unlike Xlib, though, this feature is important to Xr. Without these event handling functions, the user interface is more deeply embedded into the application code, thus reducing code portability and reusability. This is particularly inconvenient for menu handling.

A second problem is the difficulty extending Xr. An important feature of the C implementation is the ease of adding new field editors. This is made difficult in the Ada binding by the use of enumeration types to identify field editors and variant records for the field editors data structures. To add a new field editor requires modifying routines in the Ada Xr intrinsics to handle new field editors. This means an application programmer must know how to modify and recompile the Ada Xr bindings, which is unnecessary in C.

The remaining problems are more fundamental problems with the Xr toolkit rather than Ada specific problems, but these are nonetheless important in considering Xr for Ada applications. Firstly, Xr relies on the application's use of Xlib calls for some window management functions, whereas in Xt, Xlib is more transparent to the application developer. Second, the field editors have built in "look and feel". Even menus have a well-defined "look and feel" with few options. The Xr "look and feel" is fine if it agrees with the desired "look and feel" for the application, but if not, field editors written in C must be modified or be completely rewritten which becomes a problem in Ada because of the lack of extensibility. This is a serious obstacle in creating domain-tailored user interfaces.

Finally, and perhaps most importantly, Xr is a dead end. The X community has chosen Xt as the intrinsics for building toolkits. Beyond the debugging effort at Honeywell, there is no one to support or extend Xr. At an Ada X "birds-of-a-feather" session at the 1989 X Technical Conference, the major complaint of Ada X application programmers was the lack of a focal point for Ada X support. The standards community, including NIST, is writing standards based on Xt, and Ada products based on Xr will not meet future standards. Better support can be expected by staying in the mainstream of the X community. UIMSs are expected to be written to work on top of Xt, not Xr, and these UIMSs will be capable of generating Ada or C as TAE Plus does. TAE Plus currently works on top of Xr, but NASA expects to have an Xt based version in 1989, and support for Xr will probably disappear after the Xt release.

7.3 Future Directions for X and Ada

The Ada community needs to move into the mainstream of the X community, and that means moving in the Xt direction. Xr should be dropped as the Ada X toolkit, and a good Ada Xt toolkit developed. Xt is an object oriented system, and neither C nor Ada support object oriented programming. The design of Xt is based upon certain C language features such as procedure variables and weak type checking. Ada does not have these features, so an Ada implementation must find ways to provide the same functionality of Xt as the C version. Implementing the Xt intrinsics in Ada raises a number of design issues which are discussed in detail in the Appendix to this paper. An implementation is preferable to a binding because of difficulties in binding to the C data structures in a portable way.

The Ada community needs to develop a single widget set; either build one from scratch, or, if a standard widget set emerges from the X standardization efforts, use a standard widget set. For an Ada implementation of the Xt intrinsics, the widgets must also be written in Ada, thus an existing C widget set can not be used directly, but must be rewritten and kept up to date with the C versions. Finally, active evaluation of UIMSs on top of X should be pursued as this will further promote portability and reusability by separating further the user interface from application semantics. The criteria for judging UIMSs should include:

- it should be built on X
- easy access to X from an application since many UIMSs don't provide all the functionality necessary for all applications
- ease of use by programmers
- the ability to isolate application program semantics from presentation concerns
- the ability to define consistent interfaces across a wide class of tools

8 Conclusions

The X Window System is now the dominant window system in the bit-mapped graphics workstation environment. X is beginning to move into the PC market and possibly the ASCII terminal marketplace. X is the way to go for application portability, code reusability, and user interface consistency. The proliferation of X toolkits stands in the way, but industry has recognized the advantages of a single window system and toolkit, and the cooperation shown in the X Consortium, OSF and the Sun/AT&T agreement indicates the strength of the movement to a single, portable X based user interface toolkit.

The Ada community can achieve many of its portability goals by leveraging the work being done in C versions of X. The Ada community needs to develop an Xt intrinsics based version of X, and develop, either from scratch or from C implementations, a single widget set for Ada applications. Without a single widget set, portability and reusability will suffer. Of particular importance to STARS is a consistent user interface among tools in the SEE. A single Ada Xt toolkit and widget set can provide this consistency.

The Ada community must be prepared to incorporate the advances in UIMS technology as technology appears. UIMSs should be built on top of X to ensure portability, and must provide additional ease of use, reusable user interface components, and support consistent user interfaces across applications.

A An Ada/X Toolkit Interface

Clearly, Xt represents the most stable toolkit architecture, and is the beneficiary of *de facto* industry and academic consensus on application interfaces to an X toolkit layer. As mentioned earlier, for Ada to move into the mainstream of X interfaces, Xt must be considered as the basis for an Ada toolkit.

There are many obstacles to creating an Ada/X interface based upon, or derived from, Xt. Although it is claimed in [Swick 88] that Xt is "Portable across languages, computer architectures, and operating systems", even a casual study of the Xt intrinsics interfaces (or even the interface guidelines presented in [Swick 88]) will reveal that language independence has *not* been achieved. The extent to which C language dependencies have been embedded in the current Xt implementation renders impractical (or impossible) the shallow binding approach from Ada to the C interfaces, as was done by SAIC for Xrlib and Xlib.

A.1 A Language Independent Toolkit Architecture

The language sensitivity of the C Xt interfaces raises an important question: how can an Ada toolkit interface be established which leverages design, implementation, and toolkit acceptance which stems from the Xt *de facto* status? More succinctly, is it *possible* to have an Ada/X toolkit based upon Xt?

The answer to this question depends upon the extent to which Xt exhibits an identifiable language-independent architecture. It is notable that [McCormack 88] distinguishes between the Xt intrinsic mechanism and the Xt architectural model; this is an apparent separation of implementation from architecture. In our view, the Xt intrinsics are the implementation of the language independent Xt architecture, and should be examined in an effort to understand and specify what this model is, rather than form the direct basis for an Ada/X toolkit.

There is strong evidence to support this approach. Using the C intrinsics as an architectural basis for an Ada/X toolkit would introduce additional (and extraneous) complexity in the form of interface mapping of the strong Ada typing model, and the primitive (machine level) C typing model. Also, the C intrinsics implement toolkit features which are provided as native features to the Ada language. Both of these points are discussed in the following subsections.

A.2 Intrinsics Implementation of Language Features

One very interesting aspect of the Xt model is that the toolkit intrinsics layer is in fact the application program interface to the toolkit. That is, the intrinsics layer acts as an intermediary for application requests to the widget set; the application does not directly communicate with the widgets. From an Ada perspective, this seems a bit odd; consider what it would mean to create an abstraction to represent an interface to another set of abstractions. More "natural" is the idea that applications create instances of widgets, and make requests of these instances *directly*.

We can deduce reasons why Xt uses the intrinsics as the intermediary to widget objects. First, this is the only way (in C) to provide a *uniform interface* to the widget set. That is, assume widget types T1 T2 and T3 all provide a common service. The interface to these services must provide distinct entry names, since C does not support operator overloading. In C, this would result in a tight coupling of application code to particular widget implementations, which would render applications less portable. Second, the intermediary intrinsics facilitate the clean implementation of inheritance in Xt; the alternative would be to require each widget implementation to provide inheritance semantics, which would greatly increase the complexity of widget construction, and simultaneously make any toolkit less robust. Although an Xt widget programmer is responsible for specifying the inheritance hierarchy within the widget structure definition, the intrinsics actually implement the inheritance functionality.

A.3 C Interface Typing Dependencies in Xt

The Xt interfaces are also very language dependent. This has implications not only on the form of the interface, but also on the underlying implementation. That is, the projection of the C typing model on the interface would greatly constrain possible Ada implementations based upon these interfaces. There are many other instances of the projection of unchecked C programming onto the Xt interfaces. Chapter 11 of [McCormack 88], Resource Management, is notable in this regard.

Resource management allows the intrinsics to present an interface to manage resource types defined by widget programmers. Resources in Xt are named components of widgets, i.e., record fields which are addressable by string name. Thus, the resource manager provides an interface to access and manipulate fields of widgets of unknown types, whose fields are also of unknown types. To provide such generality, the Xt resource manager interface presents raw pointer types, and provides operations to perform tasks such as run-time resource type conversion.

At this time it is not clear whether this kind of interface will be necessary for an Ada/X toolkit, or whether each widget type in Ada could be responsible for its own resource management while still presenting a uniform interface. It is clear, however, that the Xt resource manager interface poses significant obstacles for a clean Ada implementation (i.e., one that exhibits a strongly typed interface).

A.4 Object Orientation: The Critical Challenge

The central theme of any Xt implementation is the provision of an object-oriented toolkit architecture. There is widespread consensus that object orientation is the appropriate window system abstraction [Goldberg 83] [Palay 88] [Young 88]. The Xt implementation was hampered in that C does not provide native language support for object oriented programming. It does, however, provide sufficient flexibility to implement the features required of object oriented languages [Stroustrup 86], e.g., procedure objects and polymorphism. Procedure pointers provide a natural mechanism for implementing *method inheritance*; type relaxation provides a natural mechanism for implementing subclass polymorphism and *resource inheritance.*

For an Ada toolkit to provide equivalent functionality and extensibility to the Xt implementation, it must exhibit (externally) the above mentioned object oriented features. Since Ada, like C, also lacks native language features to support object oriented features, most notably inheritance, it is clear that an Ada/X toolkit will also have to rely on some *intrinsic* layer. The key issue is whether Ada provides the linguistic support (e.g., procedure objects) to implement this intrinsic layer while still presenting a strongly typed toolkit interface.

A.4.1 Procedure Types in Ada

Ada does not support procedure types. This would appear to be a serious obstacle for implementing method inheritance, and for dynamic (i.e. run-time) modification of widget behaviour. The lack of procedure type in Ada was not an oversight in language design, but rather recognition that the combination of a procedure types and block scoping would create serious program reliability problems. The core of the problem is simply this: the scope of a procedure object must be identical to the scope of the procedure the object references.

Assume the existence of a procedure type in Ada, and the pre-defined prefix operation "*" used to activate the procedure object (as in C). Figure 7 illustrates in a simple setting the problem alluded to above. In this example, **nested_object** no longer exists at the time the procedure is indirectly executed. C does not suffer from this problem, since in C all functions are defined at *lexical level 0.* The point of this example is to highlight that any Ada callback implementation which makes use of procedure *pointers* will be unsafe[1]. Although Ada provides an avenue for getting the address of a procedure, and thus a (non-portable) means of having procedure objects, the above scoping problems are more serious than portability concerns, and argue strongly for an alternative mechanism for introducing procedure *types.*

One observation is that Ada does provide a task type, which could provide a basis for *simulating* procedure types. One implementation is discussed in [Lamb 83]. The problem we see with the tasking approach in general is that it is not possible to provide alternate task bodies for a particular task type, and thus each simulated procedure would represent a distinct type which has only one instance. We have experimented with unchecked conversion among task types to gain the effect of alternative task bodies for the same task entry; however, we feel this solution is inherently non-portable.

Fortunately, there is a safe procedure pointer mechanism available in Ada which makes use of Ada generics. Appendix B of this report includes the complete package specification, body, and sample application, of this mechanism. This code has been tested under two different compilers, and should be portable. The following points should be considered when examining the specification in appendix B:

[1]This is not to imply that a safe procedure type can not be designed into the Ada language

```
with text_io; use text_io;
procedure outtermost is

   global_proc: procedure;

   procedure inner_1 is

      nested_object: integer:= 999;

      procedure inner_2 is
      begin
         put(nested_object'image);
      end inner_2;

   begin
      global_proc:= inner_2;
   end inner_1;

begin
   *global_proc; -- nested object does not exist in this context
end outtermost;
```

Figure 7: Scoping Problems with Procedure Types

- This implementation is more general than the C procedure pointer mechanism, since the Ada version supports nested functions, as well as arbitrary entry points into the search path.
- The number of generic callbacks per instantiation can be varied to optimize for application needs.
- The parameter types for the callbacks can also be made generic.
- The callback package can be easily *generated* to provide for callback procedures of arbitrary parameter profiles.
- The callback implementation is free to use other (less portable but more efficient) mechanisms, e.g., the 'address solution.

A.4.2 Inheritance

The author admits to having no ready-made solution to propose for simulating inheritance in Ada (as we had a proposal for simulating procedure types). Examination of Xt provides valuable hints to one possible implementation approach, although it is debatable whether the same method would be *appropriate* for an Ada/X toolkit.

One point to note is that the Xt intrinsics do not wholly encapsulate inheritance; the widget programmer must write his widgets under an Xt discipline which will render the format of widget types (their type representation) recognizable by the intrinsics. Xt maintains a *class hierarchy*, which roughly equates to a type hierarchy. The built-in class *WidgetClass* is (directly or indirectly) a parent class of all Xt classes.

WidgetClass is known to the Xt implementation and defines common properties exhibited by all Xt widgets. Xt also has a built-in type which defines common properties exhibited by all widget *instances*, called *Widget*. The WidgetClass/Widget model corresponds to the Smalltalk model of class methods and variables being distinct from instance methods and variables.

The widget programmer who wishes to add a new widget class to Xt inserts, at the beginning of the new widget's class definition structure, the structure which defines the widget's superclass. Thus the superclass properties, or rather placeholders for these properties, are manually "compiled" into the new widget class structure.

Widget instances also contain instance data which is unique to the instance of a class. The widget programmer defines the new instance type in a similar way, by manually inserting the superclass instance structure into the new widget instance type definition. Figure 8 illustrates this structural subclassing method.

Because of this widget programming discipline, the intrinsics can perform the equivalent of an unchecked type conversion from arbitrary widget types to the base (recognized) types, Widget and WidgetClass. Thus, even though the intrinsics don't know about widget sets beyond the built-in sets, it knows how to access the relevant data fields of all widgets.

The implications of this kind of implementation on an Ada/X toolkit are unclear, although some problems are evident. First, this approach requires the intrinsics to know about representation

```
/* This structure defines the components of the core class */

typedef struct {
   WidgetClass superclass;
   String class_name;
   /* other common fields */
} CoreClassPart;

/* This structure defines the components of a subclass of the core class */

typedef struct {
   XtGeometryHandler geometry_handler;
   XtWidgetProc change_managed;
   /* other composite widget fields */
} CompositeClassPart;

/* This is the actual subclass definition, which inserts the core class */
/* components before the subclass components.                          */

typedef struct {
   CoreClassPart core_class;
   CompositeClassPart composite_class;
} CompositeClassRec, *CompositeWidgetClass;
```

Figure 8: Xt Structural Inheritance

details of widgets. This is a problem not so much because it violates the principle of information hiding as it is because it depends upon a compiler implementation choosing a physical record layout which agrees with the programmer's specified logical record layout. Of course, the widget programmer could resort to Ada representation clauses, although that too introduces portability problems [Pollack 88].

Although we propose no concrete design at this time for implementing inheritance, it appears that there are three broad avenues which should be investigated as part of an Ada/X toolkit design. These are briefly described in the following sections.

Weakly Typed Intrinsics Interfaces

This approach would project anonymous types onto the intrinsics interface, as does the Xt implementation. That is, the interface would present formal types such as *system.address*. This is an extreme approach, and one which is not likely to be recommended. The only arguments in favor of this approach would be a) necessity, if it is found that no strongly typed interface can at the same time provide toolkit extensibility, or b) that only machine-generated code would use these interfaces (e.g., UIMS-generated source).

Two points should be noted concerning the weakly-typed interface approach. First, the onus would be placed upon the application programmer to perform unchecked type conversion, or rely on the *'address* attribute, to communicate with the toolkit. Second, and more importantly, this approach would argue strongly in favor of creating a *pragma interface* binding to Xt, rather than an Ada toolkit implementation. If the toolkit needs to exhibit weakly typed interfaces in order to achieve extensibility, then mapping Ada interfaces to pre-existing (and tested) C interfaces could be the most economical solution.

Weakly and Strongly Typed Intrinsics Interfaces

This approach would distinguish two classes of intrinsics interfaces: those used by the application programmer, and those used by the widget programmer. The assumption is that the application programmer is not concerned with the *implementation* of inheritance or other toolkit extensibility functions, and so could (possibly) be insulated from these interfaces. On the other hand, the widget programmer would have to make use of such resources in order to have access to e.g., the class hierarchy which implements inheritance.

This approach also works under the assumption that strict typing works to the contrary of toolkit extensibility; if this assumption proves too pessimistic, the set of weakly-typed interfaces could be the empty set. Thus, this approach is not to be distinguished from an approach which provides a strongly typed intrinsics interface, since that would be the most desirable goal.

Program Generation Approaches

Perhaps the most extreme solution is to implement the Ada/X toolkit by means of program generators. This approach is based upon the observation that the Xt intrinsics exhibit a generalized type interface because the number and types of widgets and resources being managed are not known to the toolkit. That is, a strongly typed toolkit implementation could be constructed, but only at the expense of toolkit extensibility. However, a generator could be constructed which parameterizes this information; the toolkit interface and implementation could be *generated* from a *specification* of toolkit widgets.

This approach would depart from the Xt model, which distinguishes the intrinsics from the widget set. In the generation approach, the intrinsics would be derived from the widget set. However, it is not clear that this is a significant objection to the generation approach. In any event, widget programmers are considered as distinct from application programmers, and so the distinction between intrinsics and widget interfaces may be contrived.

A.5 Conclusions

The Xt toolkit provides the best basis for building an Ada/X toolkit which will exhibit the desired functionality and flexibility needed for domain tailorable STARS human interfaces, and will facilitate industry acceptance of the Ada/X toolkit.

Xt also presents difficult engineering problems. Most significant is the lack of a concise definition of the language independent Xt architecture. This architecture, although distinguished in [McCormack 88] from the Xt intrinsics, is nonetheless defined in terms of the Xt intrinsics. Unfortunately, the Xt intrinsics project an implementation, not an architecture, and exhibit weakly typed interfaces which would be inappropriate to mimic in Ada.

Although much work needs to be done to specify an Ada/X architecture derived from Xt, one important characteristic is clear: Ada/X must provide an object-oriented model similar to the model provided by Xt. Although we specified a mechanism for safely implementing procedure objects in the Ada/X toolkit, fundamental questions remain concerning the implementation of inheritance.

The first step in constructing an Ada/X toolkit must be specification of a language independent toolkit architecture. This architecture will be implemented by a set of language/implementation specific toolkit intrinsics. For the Ada/X toolkit, the first objective ought to be specification of clean, strongly typed intrinsic interfaces. If strong typing needs to be relaxed to provide toolkit extensibility, an attempt should be made to isolate the type relaxation to the implementation, not interfaces. Should this also prove impractical, then the weakly-typed interfaces should be isolated to interfaces used by widget programmers, and not made visible to application programmers.

Only as a last resort should alternative approaches, such as pervasive weak typing or toolkit generation approaches be examined. An important research result of the Ada/X toolkit work will be to establish the relationship between strong typing and the implementation of extensible systems.

B Complete Callback Implementation

B.1 Callback Mechanism Specification

```
package callback_mechanism is

   CALLBACK_CALL_ERROR: exception;
   CALLBACK_INSTALL_ERROR: exception;
   CALLBACK_RANGE_ERROR: exception;

   MAX_CALLBACKS: constant:= 1024;
   NUM_CALLBACKS: constant:= 3;

   subtype callback_id_range is natural range 0 .. MAX_CALLBACKS;

   -- package callback_ids provides an ADT for callback identifier types
   -- not limited private so that package callbacks can assign default
   -- null callback identifiers if no actual callback procedure is specified
   package callback_ids is
      type callback_id_type is private;
      null_id: constant callback_id_type;

      function to_callback_id_range(id: callback_id_type)
         return callback_id_range;

      private
         function next_callback_id return callback_id_range;
         type callback_id_type is record
            the_callback_id: callback_id_range:= next_callback_id;
         end record;
         null_id: constant callback_id_type:=
            (the_callback_id => callback_id_range'first);
   end callback_ids;

   use callback_ids;

   -- the default procedures will never actually be called
   procedure default_next_call_back(id: callback_id_type; s: string);
   procedure default_callback(s: string);

   generic
      -- This implementation supports three callbacks per instantiation.
      -- If the user has 4 actual procedures, two instantiations are needed.
      with procedure cb1(s: string) is default_callback;
      id1 : in callback_id_type:= null_id;

      with procedure cb2(s: string) is default_callback;
      id2 : in callback_id_type:= null_id;

      with procedure cb3(s: string) is default_callback;
      id3 : in callback_id_type:= null_id;

      -- next_callback chains callback instantiations, giving the effect
      -- of a linked list of package instantiations!
      with procedure next_callback(id: callback_id_type; s: string)
         is default_next_call_back;

      package callbacks is
         procedure callback (id : callback_id_type; s: string);
      end callbacks;

end callback_mechanism;
```

B.2 Callback Mechanism Body

```
with system; use system;
package body callback_mechanism is

   type callback_mapped_id is
      range callback_id_range'first .. callback_id_range'last * NUM_CALLBACKS;

   unmapped_id: constant callback_mapped_id:= callback_mapped_id'first;

   callback_id_map : array(callback_id_range) of callback_mapped_id:=
      (others => unmapped_id);

   starting_at: callback_mapped_id:= callback_mapped_id'first + 1;
   next_mapped_id: callback_mapped_id:= starting_at;

   package body callback_ids is

      next_id: callback_id_range:= callback_id_range'first + 1;

      -- return a unique callback id
      function next_callback_id return callback_id_range is
         i: callback_id_range:= next_id;
      begin
         next_id:= next_id + 1;
         return i;
      exception
         when constraint_error =>
            raise CALLBACK_RANGE_ERROR;
      end next_callback_id;

      -- select the callback id from the callback object
      function to_callback_id_range(id: callback_id_type)
         return callback_id_range is
      begin
         return id.the_callback_id;
      end to_callback_id_range;

   end callback_ids;


   -- these procedures should never be called, so raise exception
   procedure default_next_call_back(id: callback_id_type; s: string) is
   begin
      raise CALLBACK_CALL_ERROR;
   end;
   procedure default_callback(s: string) is
```

```
begin
   raise CALLBACK_CALL_ERROR;
end default_callback;


package body callbacks is
   -- each instantiation of callbacks has a distinct id range
   low_range, high_range: callback_mapped_id:= callback_mapped_id'last;

   procedure callback (id : callback_id_type; s: string) is
      -- subtype assignment allows use of case statement
      subtype callback_range is callback_mapped_id range 1 .. NUM_CALLBACKS;
      mapped_id: callback_mapped_id:=
         callback_id_map(to_callback_id_range(id));
      index: callback_range;
   begin
      if mapped_id in low_range .. high_range then
         index:= mapped_id - low_range + 1;
         case index is
            when 1 => cb1(s); -- call the actual callback
            when 2 => cb2(s);
            when 3 => cb3(s);
         end case;
      else -- in the range of a previous instantiation
         next_callback(id, s);
      end if;
   end callback;

begin -- initialize
   low_range:= starting_at;
   high_range:= low_range + NUM_CALLBACKS - 1;
   starting_at:= high_range + 1;

   -- do this if .. then code for each formal callback
   if cb1'address /= default_callback'address then
      if id1 /= null_id then
         if callback_id_map(to_callback_id_range(id1)) /= unmapped_id then
            raise CALLBACK_INSTALL_ERROR; -- valid procedure, duplicate id
         else
            callback_id_map(to_callback_id_range(id1)):= next_mapped_id;
         end if;
      else
         raise CALLBACK_INSTALL_ERROR; -- valid procedure, null id
      end if;
   end if;
   next_mapped_id:= next_mapped_id + 1;

   if cb2'address /= default_callback'address then
```

```
        if id2 /= null_id then
           if callback_id_map(to_callback_id_range(id2)) /= unmapped_id then
              raise CALLBACK_INSTALL_ERROR; -- valid procedure, duplicate id
           else
              callback_id_map(to_callback_id_range(id2)):= next_mapped_id;
           end if;
        else
           raise CALLBACK_INSTALL_ERROR; -- valid procedure, null id
        end if;
     end if;
     next_mapped_id:= next_mapped_id + 1;

     if cb3'address /= default_callback'address then
        if id3 /= null_id then
           if callback_id_map(to_callback_id_range(id3)) /= unmapped_id then
              raise CALLBACK_INSTALL_ERROR; -- valid procedure, duplicate id
           else
              callback_id_map(to_callback_id_range(id3)):= next_mapped_id;
           end if;
        else
           raise CALLBACK_INSTALL_ERROR; -- valid procedure, null id
        end if;
     end if;
     next_mapped_id:= next_mapped_id + 1;

  end callbacks;

end callback_mechanism;
```

B.3 Callback Mechanism Use

```
with callback_mechanism; use callback_mechanism;
with text_io; use text_io;
procedure test_callback_mechanism is
   use callback_ids;

   procedure p(s: string);
   procedure q(s: string);

   p1: callback_id_type; -- p1 and q1 now have valid callback ids
   q1: callback_id_type;

   -- in p_callbacks, cb2 and cb3 are "default" callbacks
   package p_callbacks is new callbacks(cb1 => p, id1 => p1);

   -- q_callbacks uses p_callbacks callback routine to chain instantiations
   -- procedure p and q could have both been installed in a single
   -- instantiation, but we're demonstrating instantiation chaining.
   package q_callbacks is new callbacks(
      cb1 => q,
      id1 => q1,
      next_callback => p_callbacks.callback);

   use q_callbacks; -- make the last instantiation directly visible

   -- procedures p and q do different things
   procedure p(s: string) is
   begin
      put_line("P:" & s);
   end p;
   procedure q(s: string) is
   begin
      put_line("Q:" & s);
   end q;

begin
   callback(p1, "hello world");
   callback(q1, "hello world");
end test_callback_mechanism;
```

References

[Burleigh 88]	Burleigh, David. Programming The Xt Toolkit. 1988. Xhibition'88 X Toolkit Tutorial.
[Durand 89]	Durand, Jennifer-Ann M., Young, Michal, and Troup, Dennis B. A Tool Builder's Guide to Chiron (Version 0.1, Preliminary). 1989. Arcadia Document UCI-88-18.
[Goldberg 83]	Goldberg, Adele and Robson, David. *Smalltalk-80 The Language and Its Implementation.* Addison-Wesley, 1983.
[Hanner 88]	Hanner, Mark Allen. Gambling on Window Systems. *UNIX Review* 6:50–59, December 1988.
[HewlettPackard 88]	Hewlett-Packard. Programming With the Xrlib User Interface Toolbox. 1988.
[Hinckley 89]	Hinckley, Kee. OSF/MOTIF. 1989. Presentation at the 1989 X Technical Conference.
[Institute 88]	Institute, Software Engineering. An Introduction ot the Serpent System. 1988.
[Lamb 83]	Lamb, David A. and Hilfinger, Paul N. Simulation of Procedure Variables Using Ada Tasks. *IEEE Transactions On Software Engineering* SE-9:13–15, January 1983.
[Lowgren 88]	Löwgren, Jonas. History, State and Future of User Interface Management Systems. *SIGCHI Bulletin* 20:33–44, July 1988.
[McCormack 88]	McCormack, Joel, Asente, Paul, and Swick, Ralph R. X Toolkit Intrinsics - C Language X Interface. 1988. X Version 11, Release 2.
[Morris 89]	Morris, Robert R. and Brooks, William E. A Graphic Comparison. *PC Tech Journal* 7:106–118, February 1989.
[Myers 88]	Myers, Brad A. *Tools for Creating User Interfaces: An Introduction and Survey.* Technical Report CMU-CS-88-107, Carnegie Mellon University, January 1988.
[NASA 88]	NASA. Introduction to TAE PLUS. 1988. Century Computing, Incorporated, 88-TAE-INTRO1B.
[OReilly 88]	O'Reilly and Associates. *Xlib Programming Manual for Version 11 Release 2 of The X Window System Volume One.* 1988.
[Palay 88]	Palay, Andrew J., Hansen, Fred, Kazar, Mike, Sherman, Mark, Wadlow, Maria, Neuendorffer, Thomas, Stern, Zalman, Bader, Miles, and Peter, Thom. The Andrew Toolkit - An Overview. In *Proceedings USENIX Technical Conference.* Winter 1988.
[Pollack 88]	Pollack, B. and Campbell, D. The Suitability of Ada for Communcations Protocols. In *Sixth National Conference on Ada Technology*, pages 170–181. March 1988.

[Scheifler 86] Scheifler, Robert W. and Gettys, Jim. The X Window System. *ACM Transactions on Graphics* 5:79–109, April 1986.

[Stroustrup 86] Stroustrup, Bjarne. *The C++ Programming Language.* Addison-Wesley, 1986.

[Swick 88] Swick, Ralph R. and Weissman, Terry. X Toolkit Widgets - C Language X Interface. 1988. X Version 11, Release 2.

[Young 88] Young, Michal, Taylor, Richard N., and Troup, Dennis B. Software Environment Architecture and User Interface Facilities. *IEEE Transactions on Software Engineering* 14:697–708, June 1988.